# Sublime Surf: Final Report

Karis Russell, Nathan Sfard

# 1. Introduction

Surf conditions change rapidly day to day and location to location, which forces modern day surfers to utilize online forecasts and obtain a detailed knowledge of the places they want to surf. To ease this pain, we are developing SublimeSurf, an iOS application that will keep track of the current surf conditions and allow users to rate aspects of the surf. We plan to use this rating data in combination with surf forecast data available online to notify a user when conditions look favorable, based on their previous ratings. We also intend to mine the data submitted by all users to establish generally agreed conditions that are favorable at certain locations.

## 1.1 Goals and Objectives

1. Make an iOS application to accomplish the following goals
   a. Forecast surf conditions (including aspects such as swell height/direction, wind speed/direction, tides, etc.).
   b. Allow the user to specify what they believe are favorable conditions for a specific location by selecting certain condition combinations. Allow the user to specify as many of these combinations as they want.
   c. Allow the user to rate certain default categories of their surf session as well as user-created categories (these categories will be aspects such as power, steepness, consistency, break speed, etc.).
   d. Build a set of favorable conditions based off of the category ratings specified in each session and periodically compare the computed favorable conditions to those stated by the user in Goal 1b.
   e. When a favorable combination of conditions is forecasted, notify the user.
   f. Aggregate all users' data to determine overall ratings and favorable conditions for each location.
   g. Utilize a user interface that is easy to navigate and simple to understand.
2. Gather and implement feedback from other surfers
3. Deploy the iOS application on Apple's App Store

## 1.2 Outcomes and Deliverables

- Outcomes
  - Have weekly updates with Dr. Nico
  - Utilize a backend service that works in conjunction with an iOS app
  - Create an app that is useful for surfers
  - Deploy the app onto Apple's App Store
- Deliverables
  - A demonstration of the application
  - A report detailing the process of designing, implementing, and evaluating the application
  - Source files for the application
  - App Store deployment status

# 2. Background

## 2.1 Predicting Satisfactory Surf Conditions

Firstly, these surf conditions include aspects of the surf such as swell height, direction, and period; wind speed and direction; tide stage, and much more. These conditions are only available from companies like Surfline and Magicseaweed that have a team of meteorologists that use raw, open ocean buoy data supplied by the National Oceanic and Atmospheric Agency (NOAA) to make predictions about how those open ocean conditions will affect specific locations in the future.

Although there are general patterns of what conditions are deemed satisfactory for a specific location, the term satisfactory is still subjective. In order for surfers to keep track of what specific conditions are satisfactory to them, they must go surfing often and keep a journal (whether it is a physical or mental journal) that lists the conditions present during their session and how it affected the surf.

The surf journal approach to predicting how satisfactory a forecast will be is a good approach but it requires the surfer to constantly check forecasts and maintain a surf journal. A great solution to predicting a satisfactory forecast is one that prompts the user for a time range and location of a good surf session, saves the condition information associated with that session, compares these saved conditions to forecasted conditions for a given spot, and notifies the user whenever saved conditions match forecasted conditions. This solution is great because the only thing surfers have to do is go surfing and make a journal entry whenever the surf is good and they will get notifications whenever the surf matches one of their previously good sessions. This

solution is not perfect, however, because surfers will have to go surfing for a while before they start receiving notifications and they will have to maintain a surf journal during that time.

## 2.2 Current solutions

Currently, some surf forecast apps allow users to set up alerts, which include condition combinations that the user believes to be satisfactory, and the app will notify the user whenever the forecast matches these condition combinations. These alerts are a step in the right direction but they require the user to know what conditions are satisfactory and access to this alert system is often paired with a paid subscription.

One app that we found, Glassy, is a free app that allows users to make journal entries, set up alerts, and even share journal entries online. This app represents almost everything we want our application to do, however, Glassy does not provide a connection between good sessions and alerts and the user interface is built on a cross-platform framework[1]. The cross-platform framework in this specific scenario is not designed well for an iOS device since certain gestures are not supported and the interface navigation does not match the “iOS” style of navigating from screen to screen.

# 3. Description

## 3.1 Requirements

### 3.1.1 User Interface Requirements

3.1.1.1 User shall be able to view Forecast Data:

- View forecast data for a set of saved surf spots
- Add surf spots to the saved set by...
    - Searching for spots nearby
    - Searching for spots by location/name
    - If spot is not in our database…
        - Users can create a new spot but must specify the spot’s location so that the closest spot in our database can be used for forecast data
            - Users shall have the option to pick which spot is used for this new spot’s forecast data

[1] A cross-platform framework is a framework that allows a developer to create a mobile application for any platform by simply creating a web-style application (i.e. using HTML, CSS, and Javascript).

3.1.1.2 User shall be able to view/add/edit Sessions:

- Add a surf session to their journal including the following info about the session:
  - Spot from saved spots
  - Start and end time
  - Category ratings
  - Notes
- View all past surf session entries
- Delete any surf session entry
- Edit any surf session entry

3.1.1.3 Users shall be able to view/add/edit Preferences:

- Add a specific preference combination, which includes the following optional fields:
  - A spot from the saved spots list
  - A time window
  - Swell height, direction, and/or period (range)
  - Wind direction and/or speed (range)
  - Tide stage(s)
  - Weather
  - Minimum ratings for desired condition categories
  - Whether or not to notify the user when conditions match this preference
- View all preferences
- Delete any preference
- Edit any preference

3.1.1.4 Users shall be able to access Settings:

- Create and delete condition categories
- Manage when they will be notified
- Change measurement units
- View and edit account info
- Access a help page

3.1.1.5 User interface shall be easy-to-use.

## 3.1.2 Database and Data Storage Requirements

3.1.2.1 The database shall follow a NoSQL schema

- Data shall be represented as JSON objects
- Objects shall be stored in indexed arrays

3.1.2.2. The database shall efficiently store data

- Surf spots and forecast information shall be stored as separate JSON objects with matching keys

3.1.2.3. The database must have authentication rules
- Only authenticated users may read and write to the database
- Only administrator may update forecast information

### 3.1.3 Data Retrieval Requirements

3.1.3.1 Service shall retrieve surf spot information from around the world
- This surf data shall include at minimum:
  - Spot name
  - Latitude
  - Longitude

3.1.3.2 Service shall retrieve forecast data from trusted third-party provider
- The forecast data object shall include:
  - Swell Height
  - Swell Direction
  - Swell Period
  - Wind Speed
  - Wind Direction
  - Tide

- The database shall store one week's worth of forecast information at any given time

- The database shall be updated every day at 12AM (PST)

## 3.2 Technical Problems

### 3.2.1 Remote Storage

The first technical issue discovered in our project is finding an appropriate remote storage service. There are many options for database services, but we need one that can handle a large amount of data, can integrate with iOS applications, and that will allow for mobile features such push notifications. Furthermore, neither of us has much back-end development experience, so it is overwhelming to choose only one of the seemingly great options.

### 3.2.2 Routine Service

Once our project has a storage solution in place, it is essential that we have forecast data to fill it. However, finding such forecast data is not as easy as we anticipated. Most existing surf applications and websites spend tons of time and thousands of dollars on wave forecast algorithms;therefore, they do not want to share their forecast data for free.

After finding an appropriate source of forecast data, the next problem arises because in order for the system to provide routine forecast retrieval, it must run once a day and update the database. Firebase does not allow users to run server code, so this data retrieval must be done through an external script. This causes issues because owner of the script must be authenticated to write to the database

### 3.2.3 Event Driven System

Our system is closely tied with users and a remote database, which both take a while to respond to the system[2]. If a system is responding to user requests or waiting for a remote server to respond to the system's own requests, that system must behave asynchronously. Specifically, the system must not poll for an event to occur because this wastes precious CPU time, the system must only react to registered events and release the CPU as soon as the specific event has been fully handled.

Additionally, when a particular task is attached to a request (e.g. a routine to be performed upon receiving a response) the task must be performed using the intended context surrounding the task when the original request was made. This must remain true even if the original context is no longer in use by the system.

### 3.2.4 Easy-to-Use Interface

Since this project requires making a consumer application, the application must be easy to navigate and present data clearly. These qualities are important because a user's experience is heavily influenced on how quickly and intuitively they can reach a desired outcome. Ease of use and navigation can be measured in how many UI triggers the user must perform or how intuitive the procedure was in order to reach that outcome. Presenting data clearly and intuitively in terms of position, size, and order will also improve a user's experience.

[2] A while is in terms of CPU time, which is ~2 GHz for our system.

## 3.3 System Architecture and Design

### 3.3.1 Overall System Design

The overall system in this application is composed of four major components. The first major component is the user. The user is the component that drives the requirements of this project, which specify which commands the user must be able to perform and which views the user must be able to see. Therefore, the user depends on the next major component, the frontend, to satisfy these requirements. The frontend component's core functionality is satisfying user requests to view or update data. In order to view and update data, the frontend must interact with another closely tied major component, the backend. The backend consists of a database, for use by the frontend, and a routine to populate the database with surf forecasts. In order to populate the database with surf forecasts, the backend routine must interact with the last major component of this application, Spitcast's forecast service. Spitcast is a third party surf forecaster that has a free web API for requesting surf forecasts. A flow of requests and responses propagating through the overall system can be seen in Figure 1.

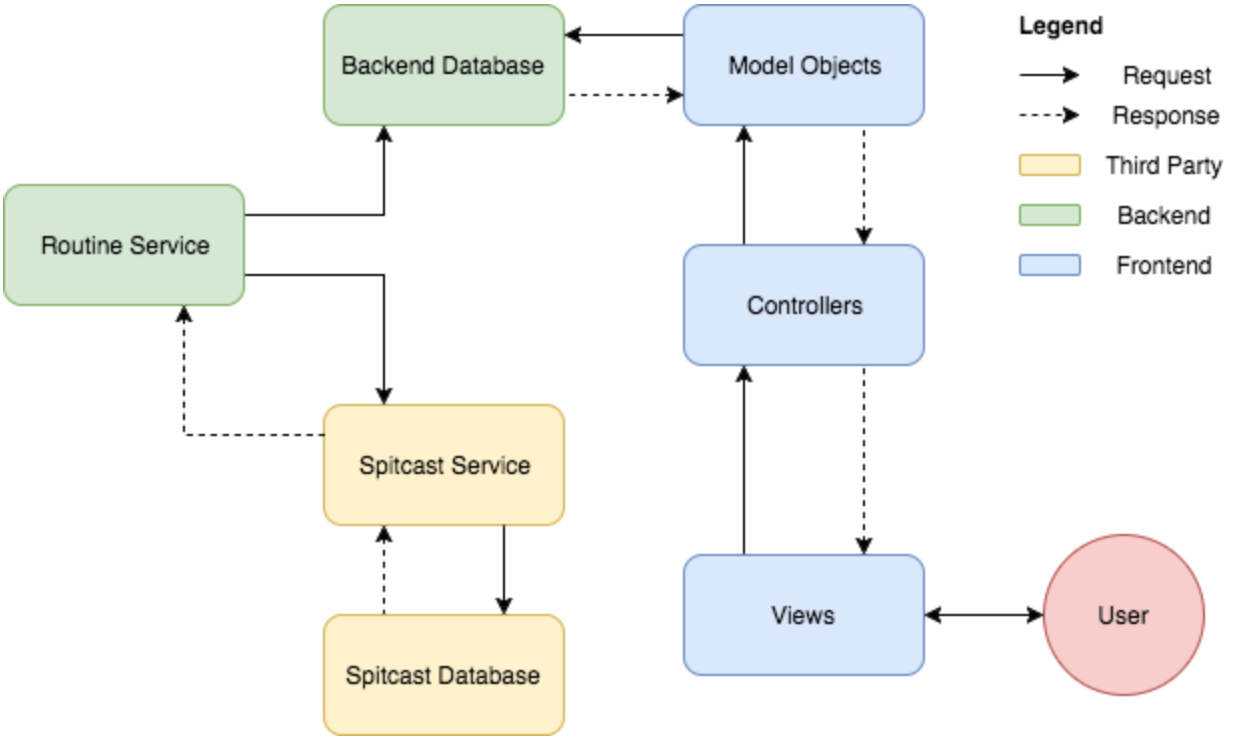


*Figure 1: Overall System Flow*

### 3.3.2 iOS Application Design

3.3.2.1 MVC Design

The Model View Controller (MVC) design enables a user targeted application to behave in an asynchronous manner. The Model represents the application's data and business logic; the Model can be reused across platforms. The View is what the user sees and interacts with; the View can be reused for different applications on the same platform. The Controller is the only piece that cannot be reused, it is what connects the Model and View to complete the system. The Controller is what creates the handlers that are given or called by either the Model or the View when the application is responding to events. The relationship between Model, View, and Controller can be seen in Figure 2.

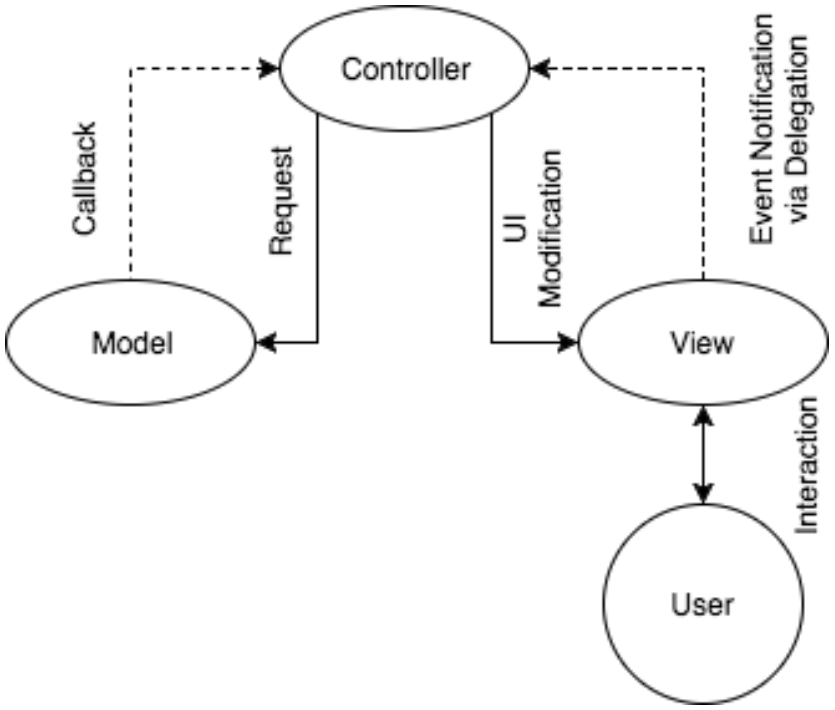


*Figure 2: Overview of the MVC Design*

3.3.2.2 Controllers

The first step to implementing an MVC design is to split up the different portions (or screens) of the application and make these all controllers. The next step is to connect these controllers to each other and to their respective Views and Models in a reasonable way. A graph of all the Controllers and their relationships can be seen in Figure 3. Each node in the directed graph in Figure 3 also represents one screen, which means that graph also represents how a user is allowed to navigate through the application.

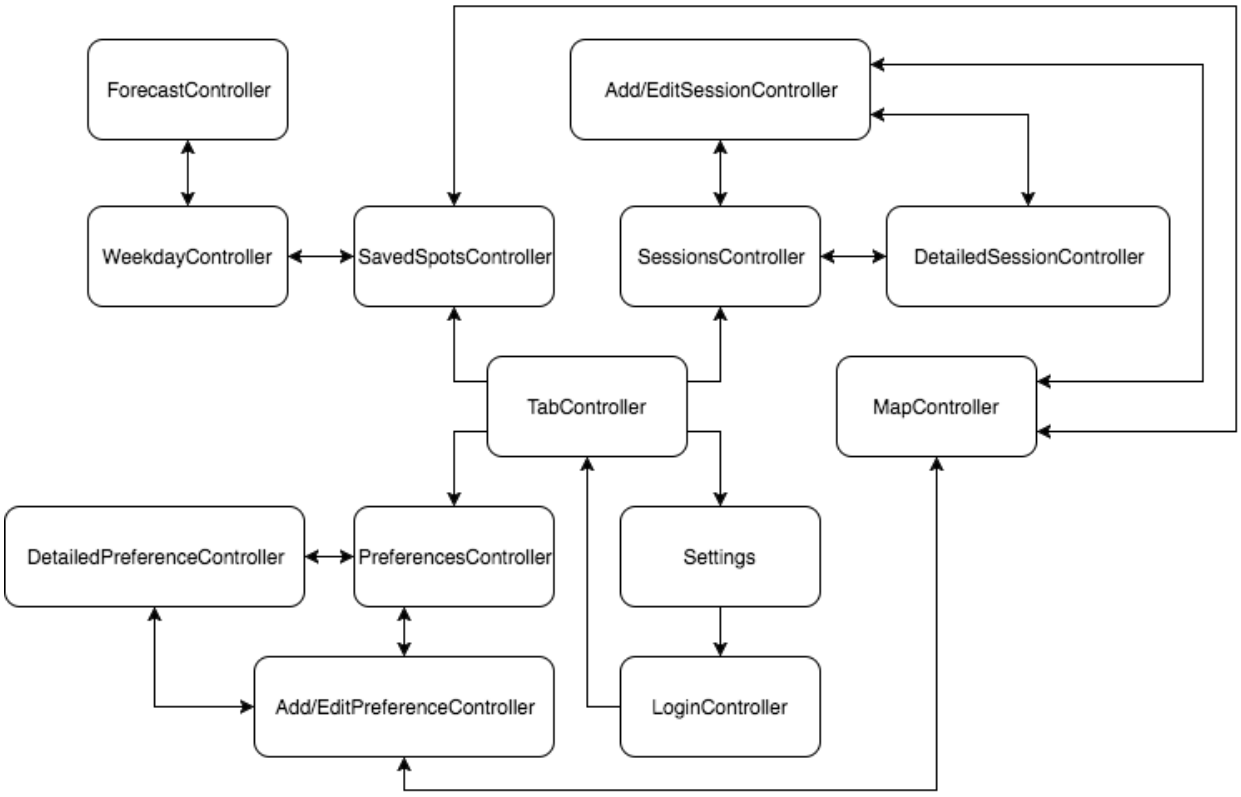


*Figure 3: Controller Relationships*

3.3.2.3 Models

Model objects take data requests from a Controller and respond by calling a given callback when the data is available. If a Controller needs a response to a certain data request on a Model, the Controller must supply an anonymous function to be run when the data is available. This callback registration is accomplished using Swift closures. Swift 3 is Apple's latest programming language at the time of this writing. One of the many convenient aspects of the language is the use of closures for asynchronous tasks. A closure is an anonymous function[3] that is passed to a request (e.g. a database fetch) and is intended to run once the request is fulfilled. Figure 2 shows how Controllers use closures as callbacks when requesting data from a Model.

Furthermore, since Swift is an Object Oriented language, closures often require properties or methods (the context) of their enclosing object. Swift's closures can retain this context when the enclosing object is no longer referenced, by saving a reference upon creation of the closure. This enables the closure to successfully and correctly execute once a specific event has occurred.

[3] An anonymous function is a function or routine that is created on the fly, usually as a parameter to another function.

Figure 4 shows the Model objects used in this application and their respective methods and properties. Note that every method accepts a closure to be called when data is available.

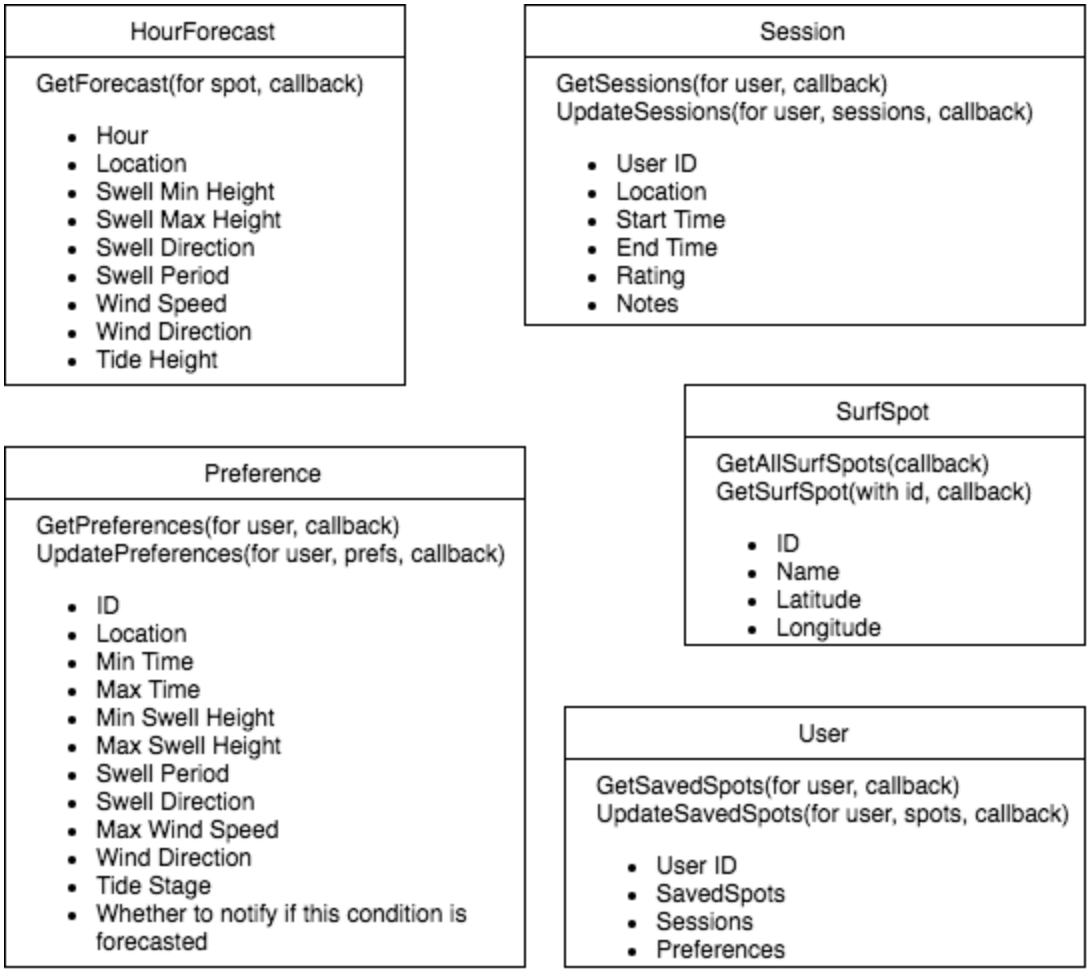


*Figure 4: Model Objects*

3.3.2.4 Views

Views are the interface between a Controller and the user. When a user interacts with a View, the View will call a specific function stored in one of its properties, called a delegate. In order to respond to these interactions, a Controller must implement these specific functions and set itself as the View's delegate. A visual representation of the relationship between View and Controller can be seen in Figure 2.

At the time of this writing, xCode 8 is Apple's latest IDE for App development. The "Interface Builder" that comes with the IDE is very useful when it comes to designing a UI and setting up these user interaction delegates. Using Interface Builder, Views are arranged and laid out in containing view controllers. These graphical view controllers are an abstraction of the Controller objects that control the Views they each contain. Interface Builder allows a developer to visually

connect a View in a storyboard file to a delegate function in the corresponding controller file, easing the process of handling user events.

### 3.3.3 Database Design

Firebase is a NoSQL database system. Therefore, the system design revolves around JSON objects. Each surf location is stored as a “spot” JSON object. These objects include additional information about the spot’s name, the name of the county it is located in,  its latitude and longitude. The Spitcast database we use for our data also assigns each spot a unique spot id. Our system leverages this spot id by using it as the unique key for each spot object, as well as each list of forecast objects. Using the spot id as the unique key for both forecast and spot objects allows the iOS application to quickly and easily query the database for forecast information corresponding to any given spot.

Each forecast object is JSON array which contains hourly-forecast JSON objects. Each forecast array begins with an object with key ‘0’. This corresponds to the hourly-forecast object for 12AM on the current day. Each successive object in the array has a key value of one greater than the previous, and contains data for the next hour in the day. These numerically increasing key values allows the application to easily iterate through the array and obtain all the forecast data.

### 3.3.4 Data Retrieval and Storage Scripts

#### 3.3.4.1 Data Source

All of spot and forecast data for the system is provided through a company called Spitcast. The Spitcast database consists of information on surf spots throughout California and forecast information for each spot. All data are stored as JSON objects and can be accessed using Spitcast’s web API.

#### 3.3.4.2 Python Storage Scripts

In order to retrieve this data from Spitcast and store the information in Firebase, the system uses two Python scripts. The scripts use Python’s URL library (urlib) to make HTTP requests through Spitcast’s web API. These scripts then utilize a Python library called Pyrebase to easily store the Spitcast data into Firebase. The first script deals only with individual surf spots, pulling geographical information about each spot as well as each spot name and county. The second script runs every twenty four hours, pulling one week’s worth of hourly forecast information for each spot.

# 4. Evaluation

## 4.1 Frontend

### 4.1.1 Model Object Unit Tests

The purpose of Model objects is to communicate with a backend database and to perform any necessary business logic, therefore, the purpose of Model object unit tests is to test the interface that connects each Model object with a Controller. We only need to test these interfacing functions and no other functions because these are the “catch-all” functions the Controller and, consequently, the user depend on to be correct. SublimeSurf performed accurately when running these unit tests, which are described in more detail below.

For data retrieval, a unit test will inject a test object into the backend database, retrieve it using the Model object’s interfacing function, and compare it to the injected object.

For data modification, a unit test will modify data, retrieve the same data, and verify that the data in the backend database has been modified.

Business logic functions include finding SurfSpots with a given id and getting a Forecast for a given SurfSpot. These are all tested similarly to the data retrieval unit test, where a known object is injected then retrieved and compared with the original.

### 4.1.2 Application Crash Tests

Applications can crash under three different circumstances: the application is being used normally, the application is being used abnormally, or the application is left running over a long period of time.

Going through each screen and invoking every UI element in every order permutation can be considered an exhaustive normal usage test. Using the application to perform every task it was intended to perform (e.g. creating/editing a session, viewing a forecast, etc.) can be considered a practical normal usage test.

Abnormal usage any usage that was not originally intended by the application’s designers. Abnormal usage can be simulated by running what experts call a  “monkey test.” A monkey test is exactly what it sounds like, invoke the test device in a random and nonsensible way for some time and see what happens. If the application is in a safe state, then it passed the monkey test.

We performed the normal usage tests and the abnormal usage tests and SublimeSurf did not crash and remained in a safe state after each test. The device was then left running in the background for 48 hours, in order to test for crashes occurring over a long period of time, and the application remained in a safe state and running the entire time.

### 4.1.3 Ease-of-Navigation Test

Often, when testing ease-of-navigation, the metric of longest navigation path is used as a test. The longest navigation path can be measured as the longest path between any two screens in an application. The controller relationships in Figure 3 can be viewed as a directed graph, in which each node represents one screen. The longest navigation path is the longest path between any two nodes on this graph. After inspection, one can find that the longest path takes is five screens.

### 4.1.4 Network Utilization Test

This mobile application will be utilizing a backend database to store data, which will require a network to transfer that data. Since non-wifi network usage is now limited by cell phone companies, it is important to measure and evaluate how much data is going to be transferred over the network.

Each user will have a number of sessions, preferences, and saved spots associated with their account. The application will download all of the user's sessions, the user's preferences, the forecasts for the user's saved spots, and all available surf spots upon application launch. When a user updates a session, preference, or saved spot, the entire list is uploaded over the network. These are the only two situations in which data is transferred over the network. Below is a table listing the data necessary to download and its corresponding estimated size[4].

| Datum | Size (bytes) |
|---|---|
| Each 7-day forecast | 101600 |
| Each surf spot (175 total) | 200 |
| Each session | 300 |
| Each preference | 300 |

[4] The estimated size is the size of the raw JSON data in bytes, rounded up to accommodate for extra overhead associated with sending data over a network.

Thus, if a user has saved X spots, Y sessions, and Z preferences, then the application will download (X * 101600) + (Y * 300) + (Z * 300) + (175 * 200) bytes on application launch. Similarly, if a user adds a session, then the application will upload (Y + 1) * 300 bytes.

# 4.2 Meeting Requirements

## 4.2.1 User Interface Requirements

4.2.1.1 Requirement Statuses Table

| **Requirement (User shall be able to…)** | **Status** | **Comments** |
|---|---|---|
| View forecast data for a set of saved surf spots | Passed Testing | Did not include weather. |
| Add/Remove surf spots to a user's saved set by... | Passed Testing | |
| Add/Edit/Remove/View a surf session to/from a user's journal including... | Passed Testing | Did not include category ratings in a session since this parameter would not be useful. |
| Add/Edit/Remove/View a preference to/from a user's journal including... | Passed Testing | Did not include Weather. |
| Access Settings | Not Implemented | This didn't seem too valuable, right now the settings page only has a logout button. |

4.2.1.2 Easy-to-Use Interface

The easy-to-navigate aspect of the app is implemented and measured using the longest path metric, which indicated that the longest path was five screens, which we consider an acceptable number. Presenting data in a clear way, however, is not implemented in our app. This is because we ran out of time, instead of presenting the data in a clear way, SublimeSurf simply lists all the data properties on the screen.

## 4.2.2 Data Storage Requirements

Verifying data storage requirements is difficult to do through explicit testing. Therefore, we believe it is best to evaluate this requirement according to the successful use of data by the application. The SublimeSurf app is successfully able to read, store, and display all required data types, so the data storage requirements have been met successfully.

### 4.2.3 Data Retrieval Requirements

To test data retrieval, we manually tested that script correctly updates the database. The manual tests consist of checking the database daily at 12AM and verifying that new data is properly posted for the current day.

## 4.3 Meeting Goals and Objectives

### 4.3.1 iOS Application Goals and Objectives

| **Goal/Objective** | **Status** | **Comments** |
|---|---|---|
| Forecast surf conditions | Passed Testing | |
| Allow users to specify favorable conditions | Passed Testing | |
| Allow users to rate categories of their surf session | Not Implemented | This goal is something that would be nice to include but it currently doesn't offer anything beneficial to predicting good surf. |
| Build a set of favorable conditions | Not Implemented | We weren't able to implement condition snapshots for a session which means we could not build a set of highly rated condition combinations |
| Notify users when a favorable condition is forecasted | Partially Implemented | Since we couldn't set up a notification function in Firebase, the user will only be notified on application startup if there are any favorable conditions in the forecast |
| Aggregate all users' data | Not Implemented | Besides not having any users, we didn't have time to consider how to use all user data for anything meaningful, since each user's data is extremely subjective |
| Utilize a user interface that's easy to navigate and simple to understand | Passed Testing | |
| Gather and implement feedback | Not Completed | We did not have time to complete the application to a point where we could release a beta for feedback |
| Deploy the app onto Apple's App Store | Not Completed | The third party we are using to populate our forecasts does not allow the application using their data to be used for commercial use and we did not pay for an Apple Developer's License so we did not deploy the app onto the App Store |

# 5. Conclusion

## 5.1 Achievements

We successfully created an iOS application that displays forecast information, utilizes user authentication to save a user's sessions and preferences, notifies users when forecasted conditions match a preference, and utilizes a backend database to store all data. We also wrote a script that will retrieve and store 7-day forecasts in the backend database.

## 5.2 Necessary Enhancements

Unfortunately, we did not meet all the goals and expectations we had hoped to meet when starting this project. Below is a list of necessary enhancements this project still needs done in order to meet our original expectations.

- Server-side code running within Firebase rather than through independent scripts

We need the retrieval script running in Firebase on a set interval in order for this app to continuously run without developer action.

- Using push notifications and Firebase Cloud Messaging (FCM)

It is also necessary for the backend to do all the checks to see if a forecast matches a user's preference so that the backend can send a notification (via FCM) to the application, which in turn sends a push notification to the OS to notify the user. This notification, in theory, should happen as soon as forecasts are updated in the backend database.

- A learning system to properly predict subjectively good conditions

Our system can take a snapshot of the conditions present during a high rated session, but it can only add that as a new preference. It would take some machine learning or artificial intelligence to properly use the high rated session data to determine which conditions a user subjectively likes.

- Better algorithm to determine what are 'similar' conditions

Right now, the algorithm used to determine similar conditions is underdeveloped. More time and research will help us create an algorithm that considers how important the variability of each condition really is.

- Condition snapshots for use with sessions

Currently, we have the ability perform condition snapshots for a given session as long as the session is recorded before midnight of that day (when the forecast will be replaced in the database). We don't actually expose this functionality because we don't store an entire history of forecast information, we believe only being able to use this functionality for a session on the same day is a limitation, and we didn't quite know how to set up rules for when a session can and can't be used for a condition snapshot.

- Weather

The forecast information we pull right now has every condition we originally wanted except weather and sunrise/sunset. Sunrise/sunset is nice to have but weather is a key factor in determining how conditions will be. For example, if the swell is pointing in the right direction at the right height and period, there's no wind, and the tide is how a user likes it, the conditions will seem perfect for that user except that it may be raining that day.

- Graphics

All necessary data is displayed in our application; however, tide and directions would be more useful in a graph, for the former, and an arrow, for the latter.

## 5.3 Lessons Learned

### 5.3.1 Karis

This project taught me how to evaluate backend requirements for a project and choose an appropriate solution. This included learning about different back-end services and the variety of solutions each service offers. After determining Firebase as the most appropriate solution for our project, I also learned how to make important database design decisions involving efficiency and scalability of data in a NoSQL database.

Furthermore, I learned how to manage a database externally through Rest APIs. This required learning how to use the Firebase and Spitcast APIs to store and retrieve data. I also learned to use the Python Pyrebase library to manipulate data to adhere to our project requirements.

### 5.3.2 Nathan

I learned how to make an iOS application using Swift 3 and xCode 8, Apple's latest tools necessary to build an iOS application. This also included learning how to use xCode's Interface Builder to create the user interface and handle user interaction. I also learned how to design an application according to the Model View Controller design scheme, which enabled me to understand how to communicate properly with a backend to maintain an asynchrounous system.

Additionally, I learned how to integrate a backend Firebase database with an iOS application. This involved supplying Firebase with the application's bundle identifier, building the application with Firebase's provided pod file, and using the FirebaseDatabase Swift 3 API. This process made me realize, despite how easy Firebase's API is, that creating an entire front and backend for an application is no easy task.

Lastly, I learned how complicated surf conditions can be and how immensely more complicated it is to forecast and correlate surf conditions. The complexity relating to forecasts was eased after deciding to use a third party for forecast information but the complexity of correlating surf conditions still remained.

# 6. References


1. *App Store Review Guidelines. Apple Inc., n.d. Web 1 June 2017.* <*https://developer.apple.com/app-store/review/guidelines/*>
2. *Firebase. Google, n.d. Web. 1 June 2017.* <*https://firebase.google.com/*>.
3. *Model-View-Controller. Apple Inc., n.d. Web 1 June 2017.* <*https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html*>
4. *Mullis, Jack. Spitcast Surf Forecast. n.d. Web. 1 June 2017.* <*www.spitcast.com*>
5. *The Swift Programming Language. Apple Inc., n.d. Web. 1 June 2017.* <*https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/index.html#//apple_ref/doc/uid/TP40014097-CH3-ID0*>